Roofline Guided Design and Analysis of a Multi-stencil CFD Solver for Multicore Performance

Bahareh Mostafazadeh[‡⋆], Ferran Marti[‡⋆], Feng Liu[†], Aparna Chandramowlishwaran[‡]

[‡]*EECS, University of California, Irvine, CA*
[†]*MAE, University of California, Irvine, CA*
[⋆]Both authors contributed equally to this work

***Abstract*—This paper presents the design and optimization of a Computational Fluid Dynamics (CFD) solver. Our solver is a structured mesh finite-volume code that simulates the fluid motion of compressible viscous flow at transonic speeds. One of the main challenges in this solver is the interaction between multi-stencils with different computational intensities and distinct memory access patterns.**

In this paper, we address the above challenge to design a high-performance multi-stencil solver on multicore systems. We use the roofline model as a guideline for choosing optimizations best suited for increasing the computational intensity and achieving a higher percentage of the machine peak performance. We show that optimizations and parallelizations combined, our solver achieves 105×, 159×, and 160× speedup compared to the baseline implementation on Intel Haswell, AMD Abu Dhabi, and Intel Broadwell respectively. We also compare against Domain Specific Languages (DSL), specifically Halide. Our hand-tuned solver outperforms code generated by Halide by up to 24×. Looking forward, we identify new opportunities to improve DSL's to bridge this gap.

***Index Terms*—Navier-Stokes equations, Computational Fluid Dynamics, Multi-stencils, Roofline performance model**

I. Introduction

Computational Fluid Dynamics (CFD) solvers are designed to understand complex physical phenomena in science and engineering applications. We consider the problem of solving Navier-Stokes equations which is a challenging multi-scale multi-physics problem that demands massive computations at extreme levels of parallelism. Explicit time-marching schemes are one of the popular numerical methods for solving the Navier-Stokes equations. These schemes are attractive because the solution at a given cell in a grid depends only on its neighboring cells (local interaction), and are commonly known to exhibit a stencil pattern.

Although there is a significant body of work on implementing stencil computations on both CPUs and GPUs [4], [5], [10], [22], nearly all of these studies seek to optimize single stencils. On the contrary, we look at the entire solver. Our solver is a realistic structured mesh finite-volume code and presents many challenges of real applications that are not addressed by similar papers in the literature. Specifically, our solver consists of hybrid-stencils with different computational intensities and memory access patterns. While the challenges of single-stencil kernels still apply to this problem, solving a hybrid multi-stencil kernel poses new challenges, since each stencil has a unique pattern and requires access to different neighbors. As a result, finding the optimal schedule to balance redundant computation, locality, and parallelism is non-trivial.

First, we begin by characterizing the different stencil patterns in the solver. Second, we systematically outline the different optimization techniques guided by the roofline performance model [24]. We analyze how multiple stencils impact optimization and discuss techniques for improving locality and parallelism by trading off redundant work. Third, we ask whether CFD applications can be expressed in stencil domain-specific languages (DSLs). Specifically, we consider Halide [15] which is a popular DSL adopted by industry and academia. Finally, we also ask whether such an implementation can deliver a sufficient combination of optimizations to compete with a hand-tuned code and what are its limitations. Our case study for evaluation is an external flow solution around a cylinder that predicts a circulation bubble.

Contributions and findings. This paper makes the following contributions.

- We create a high-performance multi-stencil solver for compressible viscous flows for multicore systems. This is achieved by various optimizations such as – (1) loop re-structuring and fusion that combines different stencil sweeps and flux calculations, (2) a two-level blocking strategy for iteration order within each stencil sweep and for multicore parallelization, (3) SIMD-aware code and data layout re-structuring to enable auto-vectorization, (4) management of NUMA page placement using first-touch initialization policy, (5) eliminating false sharing using padding of data and modifying the data structure to minimize shared data, and (6) strength reduction.
- We document a systematic process guided by the roofline performance model in transforming a memory bound solver to a highly tuned one. We explain this process in a way that is useful for other tuning practitioners.
- We present an external flow solution around a cylinder as a case study to evaluate our solver at subsonic speeds and discuss the performance results. Our optimized solver achieves 105×, 159×, and 160× speedup compared to the baseline implementation [11] on Intel Haswell, AMD Abu Dhabi, and Intel Broadwell respectively.
- Finally, we implement our solver in Halide and show that it's possible for a DSL to capture realistic use cases

like this solver. However, our hand-tuned implementation exceeds code generated by Halide by up to 24×. The main performance difference is due to Halide's inability to generate efficient vectorized code and lack of support for NUMA machines. Additionally, Halide and other DSL's mainly focus on cell-centered stencils which do not capture all the stencils patterns in real applications. Our findings quantify the gap between a hand-tuned implementation and code generated by a DSL for this class of applications.

II. Solver overview and governing equations

In this section. we first describe the Navier-Stokes equations and the numerical schemes used for simulating the realistic geometries of flows. Then, we characterize in detail the different stencil patterns and their performance challenges.

A. Governing Equations

Variable	Description
$\vec{F}_{inv}$	Inviscid fluxes
$\vec{D}$	Fluxes of artificial dissipation
$\vec{F}_c$	Convective fluxes
$\vec{F}_v$	Viscous fluxes
$\vec{W}$	Conservative variables
$\vec{R}$	Residuals
Ω	Cell volume
S	Face surface
$\vec{n}$	Face normal vector
Δt	Real time step
Δt^*	Pseudo time step
α	Coefficient for Runge-Kutta scheme
$\hat{\Lambda}^S$	Spectral radii of convective flux Jacobian
ϵ	Artificial dissipation coefficients
u	Component of velocity vector in the x direction

TABLE I: Notation.

This paper is based on the laminar implementation of an existing time accurate three-dimensional Navier-Stokes code known as ParCAE [11], [26]. It solves the 3D Unsteady Reynolds Averaged Navier-Stokes (URANS) equations on structured grids using a cell centered finite-volume method. A time-accurate solution is obtained through a dual-time stepping scheme proposed by Jameson [8]. The notation that will be used throughout the rest of the paper in summarized in Table I. The set of 5 governing equations forming the Navier-Stokes equations can be written as follows.

$$\frac{d(\vec{W}\Omega)_{i,j,k}}{dt} = -\vec{R}_{i,j,k}$$

where $\vec{W}_{i,j,k}$ is the vector of conservative variables (conservation of mass, momentum in each direction, and energy) for cell (i,j,k), Ω is the cell volume, and $\vec{R}$ is the vector of residuals.

The time derivative is discretized by a backward-difference scheme of second-order accuracy as shown below.

$$\frac{3(\vec{W}\Omega)^{n+1}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t} = -\vec{R}_{i,j,k}(\vec{W}^{n+1})$$

At each time step the problem is reformulated as the following steady-state problem in pseudo time t^*.

$$\frac{d(\vec{W}^*\Omega)^{n+1}_{i,j,k}}{dt^*} = -\vec{R}^*_{i,j,k}(\vec{W}^*)$$

where $\vec{W}^* = \vec{W}^{n+1}$ when the solution is converged, and

$$\vec{R}^*_{i,j,k}(\vec{W}^*) = \vec{R}_{i,j,k}(\vec{W}^*) + \frac{3(\vec{W}\Omega)^{*}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t}.$$

For each real time step, the set of equations are marched to a steady state in pseudo time. The pseudo time marching is implemented as a sequence of pseudo time steps, each of them discretized with a multi-stage Runge-Kutta scheme. The solution at stage m is

$$\vec{W}^m_{i,j,k} = \vec{W}^0_{i,j,k} - \frac{\alpha_k \Delta t^*_{i,j,k}}{\Omega^*_{i,j,k}} \left[1 + \frac{3\alpha_k \Delta t^*_{i,j,k}}{2\Delta t}\right]^{-1} * \left[\vec{R}_{i,j,k}(\vec{W}^{m-1}) + \frac{3(\vec{W}\Omega)^{0}_{i,j,k} - 4(\vec{W}\Omega)^{n}_{i,j,k} + (\vec{W}\Omega)^{n-1}_{i,j,k}}{2\Delta t}\right] \quad (1)$$

where α is the Runge-Kutta coefficient for the corresponding stage.

The spatial discretization of the residual is 2nd order accurate and is composed of the summation of the face fluxes.

$$\vec{R}_{i,j,k}(\vec{W}) = \sum_{m=1}^{N_F} [(\vec{F}_c - \vec{F}_v)_m \vec{n}_m S_m]_{i,j,k}$$

with N_F equal to the number of cell faces (6 for a 3D hexahedral grid), S the face surface, n the normal vector to the face, and $\vec{F}_c$ and $\vec{F}_v$ are the convective and viscous fluxes accounting for inertial and viscous effects, respectively.

Defining the conservative variables at a face (e.g. at the face between cell i and cell $i+1$) as

$$\vec{W}_{i+1/2,j,k} = \frac{1}{2}(W_{i,j,k} + W_{i+1,j,k})$$

and avoiding the j,k notation, the contribution to the residual of the convective fluxes at face $i+1/2$ is

$$[\vec{F}_c \vec{n} S]_{i+1/2} \approx \vec{F}_{inv}(\vec{W}_{i+1/2})(\vec{n}S)_{i+1/2} - \vec{D}_{i+1/2}$$

where the inviscid fluxes $\vec{F}_{inv}$ are discretized with 2nd order central difference. To avoid numerical instabilities, local artificial dissipation is added to $\vec{D}$ following the classic JST scheme [9].

$$\vec{D}_{i+1/2} = \hat{\Lambda}^S_{i+1/2} \left[\epsilon^{(2)}_{i+1/2}(\vec{W}_{i+1} - \vec{W}_i) - \epsilon^{(4)}_{i+1/2}(\vec{W}_{i+2} - 3\vec{W}_{i+1} + 3\vec{W}_i - \vec{W}_{i-1})\right] \quad (2)$$

The contribution of the viscous fluxes is more challenging to calculate since it requires an auxiliary grid. The formulation in the viscous terms contains gradients of velocity. Using Green's theorem, and defining an auxiliary grid centered on the vertices

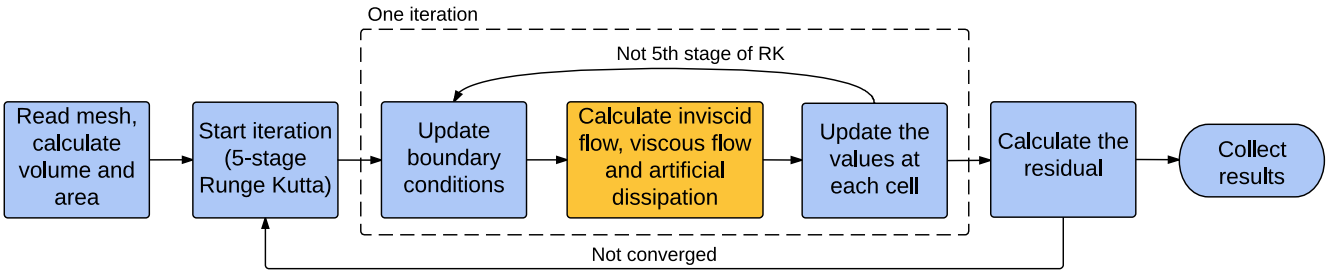

Fig. 1: High-level overview of the URANS solver. The dashed box denotes one iteration where we solve for $\vec{W}_{i,j,k}$. The yellow box highlights the calculation of the fluxes which is the computational core of the solver.

of the original grid, the gradients of velocity at such vertices are defined as (e.g. for $\partial u/\partial x$)

$$\left(\frac{\partial u}{\partial x}\right) \approx \frac{1}{\Omega_{aux}} \sum_{m=1}^{N_F} (u\, n_{x_{aux}}\, S_{aux})_m$$

where the summation is on the faces of the auxiliary grid. After computing vertex velocity gradients using the above equation, the face values of the viscous fluxes on the original grid are recovered by averaging the obtained vertex values.

B. Stencil Patterns

The overall structure of the solver is shown in Figure 1. We solve for the updated values of $\vec{W}_{i,j,k}$ in Equation 1 at each stage of the Runge-Kutta scheme which is shown as the dashed box in the figure. The yellow box denotes the calculation of multiple fluxes which is the computational core of the solver and accounts for more than 90% of the overall execution time. This consists mainly of three flux calculations namely, viscous flux, inviscid flux, and artificial dissipation. Since our scheme is explicit, each cell is updated based on the values of a subset of the neighboring cells and thus the computational pattern is a stencil. While all the flux calculations have a stencil pattern, each has a unique pattern, with a different size and a distinct memory access pattern. As we will further elaborate in this section, these multiple stencil patterns pose the main challenge in the design and optimization of this CFD solver.

We divide the stencils into two categories namely, cell-centered and vertex-centered.

Cell-centered stencils – Artificial dissipation, $\vec{D}$ and inviscid fluxes, $\vec{F}_{inv}$ are examples of cell-centered stencils. The artificial dissipation calculated in Equation 2 is a blend of second order and fourth order differences. The top of Figure 2 illustrates the stencil pattern for computing the orange colored cell. The artificial dissipation at each surface, which is shown with a red arrow, is computed using the values of the current cell (orange) and the neighbors shown in dark blue. For each direction, we compute the outgoing flux at cell (i,j,k), then reuse the values calculated at the cells $(i-1,j,k)$, $(i,j-1,k)$ and $(i,j,k-1)$ for the incoming fluxes. The stencil pattern for inviscid fluxes is similar to artificial dissipation but it only requires one neighboring cell in each direction.

Vertex-centered stencils – The viscous flux calculation, $\vec{F}_v$ is an example of vertex-centered stencils. As discussed in Section II-A, to compute the viscous fluxes we have to first find the velocity gradients at the vertices of each cell. This is shown in the bottom of Figure 2, where the colored cubes are the cells in the original grid, whereas the dashed cubes represent the cells of the auxiliary grid. The red point is the center of the auxiliary grid which is the vertex of the original grid. The viscous flux calculation consists of two stages. First, we use an 8 point stencil to calculate the gradients at each vertex (8 vertices in 3D), and then a 4 point stencil to find the flux value at each surface (6 surfaces in 3D).

Now, we can analyze the differences between cell-centered and vertex-centered stencils using the above examples. First, vertex-centered stencils require an additional sweep over the entire grid since it's a 2-stage calculation. Moreover, the 2-stages consist of 2 different stencils operating on distinct pieces of data. Second, cell-centered stencils access an equal number of neighboring cells in each dimension, whereas the memory access pattern of the vertex-centered stencils requires accessing a block of neighbors. Third, since the grid is stored in memory such that accesses in the i direction are unit-stride, accesses in the j and k directions are more expensive and vertex-centered stencils access more neighboring cells in j and k dimensions with different strides thus resulting in a more memory-bound stencil pattern compared to that of the cell-centered stencils.

Another aspect of these stencil computations that adds to the challenges of designing an efficient multi-stencil solver is that each of the stencils discussed above consists of multiple terms (such as the spectral Radii ($\hat{\Lambda}^S$), average and auxiliary surfaces, artificial dissipation coefficients (ϵ), etc.) that are used to compute the final flux value at the surface. Finding the optimal schedule for these intermediate values is non-trivial. For instance, intermediate values that recur in different stencils can either be stored in memory or computed on the fly. While the former avoids redundant computation, it causes more memory accesses. Thus achieving best performance requires finding an optimal balance of different potential trade-offs.

III. Experimental Setup

In this section, we describe the different architectures and case study used to explore our solver's performance.

Architectures – We summarize the key differences among the three multi-core SMP's used in this study: dual-socket 8-

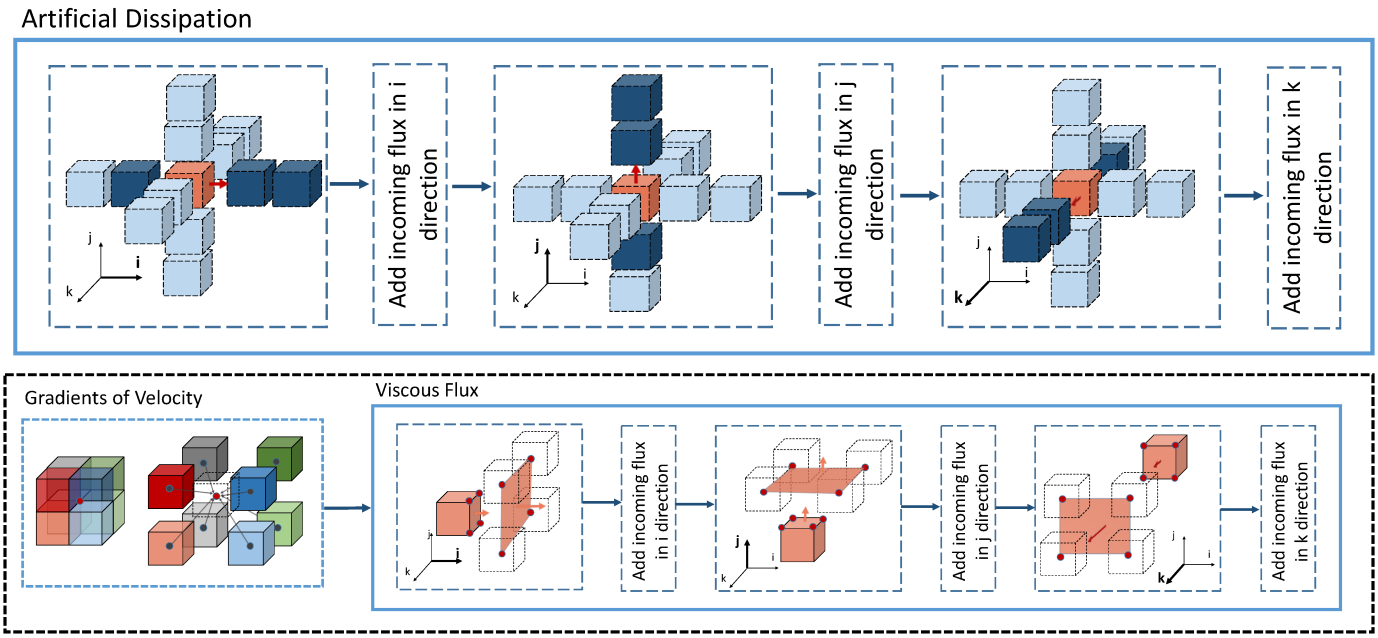

Fig. 2: Different stencil patterns of this solver, where the blue arrows show the dependency between different stages of computation. The top figure shows the stencil pattern for computation of the artificial dissipation, where the outgoing fluxes are computed for each cell and reused as incoming fluxes for neighboring cells. The bottom figure is the computation of the viscous flux, where we first compute the gradients of velocity at the center of the auxiliary grid (shown as dashed boxes), and use these gradients to compute the viscous fluxes at the center of the original grid (orange cell).

core Intel Xeon (Haswell), quad-socket 16-core AMD Opteron (Abu Dhabi), and dual-socket 22-core Intel Xeon (Broadwell). Haswell and Broadwell also support simultaneous multi-threading which enables 2 threads per core. The key parameters of these platforms are summarized in Table II.

The 3 systems have varying peak floating-point performance with Haswell at the lower end of the spectrum. Abu Dhabi and Broadwell have 1.9–2.5× higher performance than Haswell due to a large number of cores per socket. SIMD (Haswell and Broadwell support AVX2 instruction set while Abu Dhabi supports AVX instructions) enables up to 4 flops per cycle per core in double-precision (DP). Since our solver is memory-bound, we don't expect SIMD to initially improve performance.

Broadwell has a much larger L3 cache and higher peak DRAM bandwidth per socket. However, Broadwell also has a smaller DRAM bandwidth per core due to 22-cores per socket. This should enable better performance on kernels with large working sets and high computational intensity. The stencils in our solver have low computational intensity but the memory access pattern exhibits high locality. Therefore, the ultimate effects are not entirely clear a priori. Finally, while the Haswell and Broadwell systems have 2 CPU sockets, the Abu Dhabi system has 4 sockets. Therefore, we expect NUMA-aware data placement to be critical on the AMD machine for scalability.

Case study (Cylinder flow) – The case study used in this paper for performance and correctness experiments is an external flow problem around a cylinder for a grid of size 2048×1000 which results in 2 million grid points. Far field boundary conditions are implemented for the outer boundaries at $j_{\max}$. A simulation with Reynolds number of 50 and Mach number of 0.2 generates the steady solution shown in Figure 3, where circulation bubbles are formed behind the cylinder.

Architecture	Intel XE5-2630 v3 (Haswell)	AMD Opteron 6376 (Abu Dhabi)	Intel XE5-2699 v4 (Broadwell)
Frequency	2.4 GHz	2.3 GHz	2.2 GHz
Sockets	2	4	2
Cores/Socket	8	16	22
Threads/Core	2	1	2
GFlop/s (DP, SP)	614.4 1228.8	1177.6 2355.2	1548.8 3097.6
SIMD(DP, SP)	4-way 8-way	4-way 8-way	4-way 8-way
L1/L2/L3 cache	32/256/20480† KB	16/1024/16384† KB	32/256/56320† KB
DRAM Bandwidth	59.71 GB/s	51.2 GB/s	59.71 GB/s
Stream Bandwidth	102 GB/s	160 GB/s	100 GB/s
Compiler	icpc 17.0.4	icpc 15.0.3	icpc 15.0.3

TABLE II: Architectural Parameters. Note that peak DRAM bandwidth is per socket and STREAM bandwidth is for the entire node. †shared among cores on a socket.

IV. Roofline Analysis and Optimizations

Roofline [24] is a visual performance model to bound the performance of an application as a function of machine peak performance, DRAM bandwidth, and arithmetic intensity (measured as the ratio of total floating point operations to total data movement). We start by building the visual roofline model as shown in Figure 4 for each of the three systems in Table II. We use STREAM bandwidth instead of DRAM pin bandwidth to obtain a realistic roofline. Note that the flop to byte ratio at the *ridge point*, where the peak floating point performance roof meets the peak stream bandwidth roof is 6.0, 7.3, and 15.5 respectively on the three systems. As a result, we expect the code to progressively become more memory-bound as we move from Haswell to Broadwell.

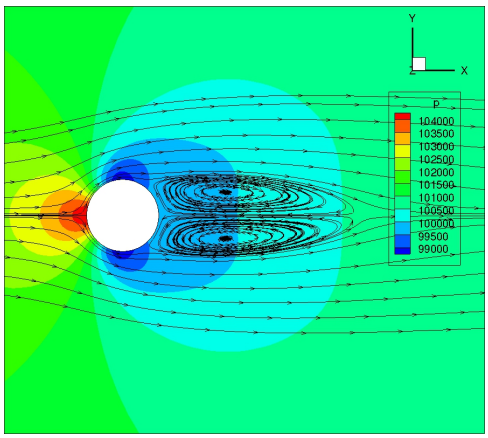

Fig. 3: Streamlines and pressure contours for cylinder simulation with Reynolds number = 50 and Mach number = 0.2. Two symmetric circulation bubbles are formed behind the cylinder.

Next, we use this model to determine the limiting factors of performance and guide the optimizations starting from the baseline code. The roofline model requires an estimate of the total floating point operations and data movement. To estimate the number of floating point operations, we use PAPI [12] and also verify the numbers by comparing it against Intel's Software Development Emulator (SDE) and *likwid* [21]. For estimating the number of bytes accessed, we use *likwid*, which uses uncore hardware counters for DRAM accesses.

Our code is based on a time accurate 3D Navier-Stokes solver written in Fortran [11] that is currently used by computational scientists. It was written with the focus of optimal computation, which is true of most codes designed during its time. As a first step, we port the code to C++. From now on, we will refer to this version as *Baseline*. The overall structure of the solver is shown in Figure 1, and the computational pattern is illustrated in Figure 2 and discussed in Section II-B. In the rest of this section, we discuss each optimization in detail and report the achieved performance and speedup on top of the *Baseline* in Figures 4 and 5 respectively.

A. Strength reduction

Math operations such as `pow` and `sqrt` were one of the hotspots observed with Intel VTune [16] in the double precision *Baseline*. Unfortunately, these instructions have long latencies and are not pipelined, thus limiting performance. For instance, on Haswell and Broadwell, double precision `sqrt` has 19-35 cycles latency[1]. To address this deficiency, we replace the scalar `sqrt/pow` with multiplication and addition [3]. Doing so requires more floating point instructions but they have low latencies and are pipelined. Apart from round-off error due to a different combination of instructions, there is no loss of overall accuracy since all the operations are done in double precision. As such, with sufficient instruction- and data-level parallelism, we expect this approach to improve performance. Figure 5 shows that strength reduction improves single-core performance by $1.2\times$, $1.4\times$, and $1.3\times$ on Haswell, Abu Dhabi, and Broadwell respectively.

[1]https://software.intel.com/sites/landingpage/IntrinsicsGuide

B. Stencil Fusion

The stencils in our solver discussed in Section II-B have dependencies between themselves (intra-stencil) and across multiple stencils (inter-stencils). Additionally, there are many valid schedules for computing the multi-stencils which have different trade-offs between redundant computation, locality, and parallelism. To improve the performance of the solver, we need to find the best schedule respecting these stencil dependencies. Below, we discuss the different trade-offs for finding the optimal schedule and propose optimizations suitable for current x86 platforms.

a) Intra-Stencil Fusion: The fluxes include two components – incoming flux and outgoing flux. In the *Baseline*, we first calculate the outgoing fluxes at cell (i, j, k) as shown in Figure 2, then reuse the values calculated at the cells $(i+1, j, k)$, $(i, j+1, k)$ and $(i, j, k+1)$ for the incoming fluxes. This is computationally efficient since each flux is calculated only once. However, the roofline plot shows that the solver is currently highly memory-bound on all 3 systems. Given this observation, trading off memory accesses for redundant computation seems like a reasonable optimization.

Therefore, instead of only computing the outgoing fluxes at each cell, we compute all six fluxes (two in each direction) for every cell, thus eliminating the need for reading the incoming fluxes from memory. Referring back to Figure 2, this optimization is fusing all the stages within each blue box into one stencil resulting in a 13-point stencil for the artificial dissipation and a 7-point stencil for the inviscid flux calculation. Although this optimization results in the redundant computation of the fluxes, it allows for improved locality and reduces the number of memory accesses. Moreover, it also removes the dependency between neighboring cells in any given iteration which enables cells to do the computation independent of one another which is better suited for parallelism.

b) Inter-Stencil Fusion: The computation of the viscous fluxes requires the gradients of velocity which are computed on the cell vertices which form the centers of an auxiliary grid. The *Baseline* performs this calculation in two stages as discussed in Section II-B where we store the values of gradients in memory that is later accessed in the second traversal to compute the viscous fluxes. The two stages comprising of the two traversals form different stencil patterns where every value is computed only once for computational efficiency. Given the memory-bound nature of our solver observed by the roofline, we fuse the two distinct stencils into a single stencil that computes the final value of the viscous flux on the surface in one traversal. The dashed black box in the bottom of Figure 2 corresponds to the stencils that are fused.

While inter-stencil fusion is worthwhile since it eliminates the memory accesses required to read the gradients from memory, it calls for redundant computation as each gradient is now computed by each of the 8 cells in 3D adjacent to that vertex. Figure 4 shows that fusion increases the arithmetic intensity

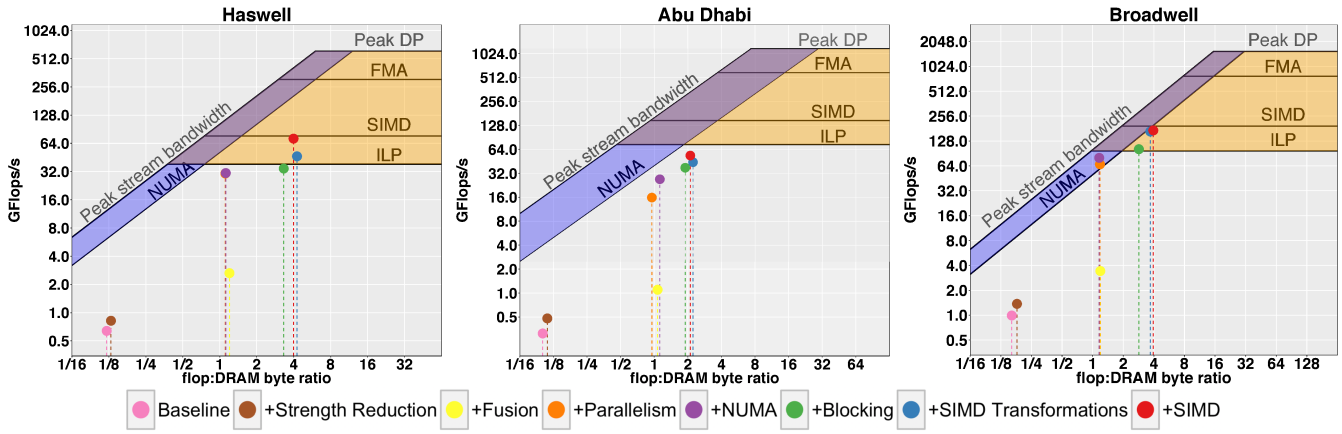

Fig. 4: Visual roofline of the three systems in Table II. The circle denotes the achieved performance (in Gflops/s) with each optimization and the dashed line shows the corresponding arithmetic intensity (flop to byte ratio).

considerably on all 3 systems from 0.13, 0.18, and 0.11 to 1.2, 1.2 and 1.1 respectively. The yellow bar in Figure 5 shows the corresponding single-core performance gain. Overall, we achieve 3×, 2.1× and 2.3× speedup from both intra- and inter-stencil fusion on top of strength reduction on the 3 respective systems. Note that since most stencil optimization studies focus only on single stencils, inter-stencil fusion is not considered and as a result, stencil-based frameworks also fail to perform this optimization. We discuss this limitation in further detail in Section V when we compare against DSL's.

C. Parallelization

Next, we parallelize the entire solver using OpenMP. We apply grid-block parallelization for all the stages of the solver including flux calculations. That is, we divide the entire grid into blocks of equal size and assign each block to a thread to exploit parallelism within each stencil. This block parallelization strategy exploits the iteration order and spatial locality within each stencil sweep. Threads are assigned first to multiple cores before multiple sockets, and multiple sockets before simultaneous multithreading (SMT). For clarity, we highlight the SMT and NUMA regions explicitly in Figure 5. Since all threads are working on blocks of equal size, there is no load imbalance.

Parallelization reduces the arithmetic intensity marginally as seen in Figure 4. This is due to an increase in the number of DRAM bytes accessed because the halos for each block are redundantly accessed by different threads. The orange bars in Figure 5 show the speedup due to parallelization on top of strength reduction and fusion. Observe that Haswell and Broadwell deliver scalability of 10.2× and 19.5× up to 16 and 44 cores respectively at which point, HyperThreading only improves performance marginally. Abu Dhabi shows good scalability of 10.8× up to 16 cores (1 socket) but little thereafter. This observation is not surprising since Abu Dhabi has 4 sockets (4 NUMA regions) and we expect poor scaling beyond 1 socket if we don't exploit NUMA-aware data allocation.

Exploiting multicore is challenging as threads share multiple resources on a chip such as caches, bandwidth, and even floating-point units. Below, we describe optimizations to improve the scalability of the solver, especially for a large number of threads and NUMA machines.

a) Eliminating False Sharing: It is essential to eliminate false sharing among multiple threads in order to achieve scalability, especially with increasing number of threads. One way to mitigate the effects of false sharing is to minimize shared data structures. In order to do so, we modify the structure of the code, so that instead of storing the different fluxes (e.g. viscous flux, inviscid flux, etc.) for the entire grid, we store them for each block separately. Therefore each thread only accesses its own block while performing the calculations and only writes the final values to a shared data structure.

Unfortunately, we are unable to eliminate all shared data. For instance, conservative variables (components of W) are updated after each stage of the Runge-Kutta scheme and the new values are used in the next stage. Re-structuring such data into individual thread blocks results in increased cost of communication among the different blocks in order to maintain the correct value in the halo region. In such cases, we pad the arrays to a multiple of the cache line size for each thread block, thereby eliminating the false sharing problem.

b) NUMA-aware Allocation: Modern SMPs are based on non-uniform memory access. All 3 systems in our study are NUMA machines. Our observations based on the roofline model implies that we will see even more performance problems on NUMA machines since data placement will be critical on a NUMA platform. This is highlighted in Figure 4 by the *NUMA* diagonal line that represents the lower bandwidth and resulting lower attainable performance.

To break this *NUMA* ceiling, we use the first-touch page allocation policy of the operating system to also parallelize the data initialization loops. We use the same domain decomposition for both the initialization and the computation loops. Consequently, each thread initializes the segment of data that it computes on and the data is initialized on its local DRAM. The purple bar in Figure 5 shows the performance gain due to NUMA-aware data allocation. The impact of NUMA-aware allocation is particularly evident in Abu Dhabi which achieves

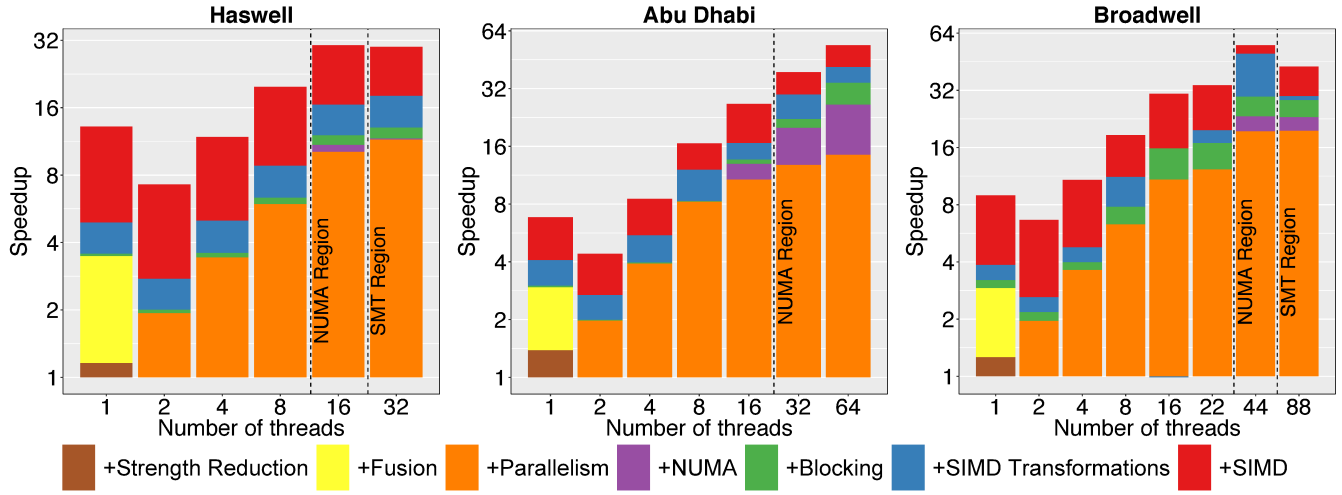

Fig. 5: Speedup achieved with each optimization for varying number of threads on the three systems in Table II. Note that the speedup for the parallel case (2+ threads) is reported on top of strength reduction and fusion optimizations.

Variable	Description	Size
$\vec{F}_{inv}$	Inviscid fluxes	Grid size $\times$ 5
$\vec{D}$	Fluxes of artificial dissipation	Grid size $\times$ 5
$\vec{F}_v$	Viscous fluxes	Grid size $\times$ 5
$\vec{W}$	Conservative variables	Grid size $\times$ 5
Ω	Cell volume	Grid size
S	Face surface	Grid size $\times$ 6
Δt^*	Pseudo time step	Grid size

TABLE III: Sizes of variables used in the solver. Note that multiplication by 5 is because of the five conservative variables and multiplication by 6 is because the surface values are stored for faces in each of the three dimensions, and each face consists of components in three directions.

an additional $1.8\times$ speedup on 4 sockets.

D. Blocking

Cache blocking is a well-known optimization for memory-bound applications to exploit spatial locality and this is especially true for stencils [4], [5], [10], [17]. This is even more critical in our solver which consists of multi-stencils. This is because we need to store several variables for each cell as opposed to single stencils that cause significant memory traffic at every stage of the numerical scheme. These variables and their sizes required for storage are summarized in Table III.

To efficiently utilize the cache, we decompose the grid into blocks and run an entire iteration (all 5-stages of the Runga-Kutta scheme shown in Figure 1) before synchronization. This introduces error in the halo regions. However, since ours is an iterative solver, the error is damped out by performing a small number of extra iterations. This decomposition is illustrated in Figure 6 for a 2D grid. We divide the grid into yellow blocks of size $LL_X \times LL_Y$ in 2D in such way that each block fits in the last level of the cache (L3 cache). Furthermore, our implementation needs to account for different stencil patterns since the data structures accessed by these stencils differs and they achieve the best performance with different sizes of $LL_X \times LL_Y$. We tune for the best block size empirically on all three systems.

Cache blocking further increases the arithmetic intensity of the solver to 3.3, 1.9, and 2.9 respectively as seen from Figure 4. Now, the solver is limited by the compute ceiling and we expect optimizations such as vectorization and increased instruction-level parallelism to further improve performance.

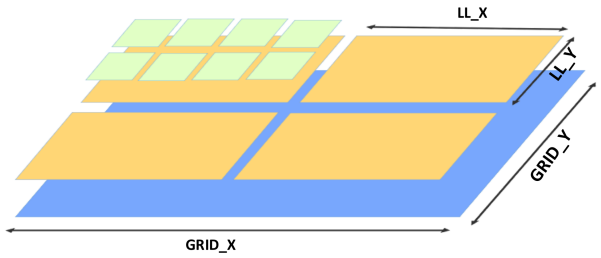

Fig. 6: A two-level blocking strategy for locality and parallelization. The yellow cache block is tuned such that the data used in a block of size $LL_X \times LL_Y$ fits into the last-level cache and each green block is assigned to one thread.

E. Vectorization

In x86 architectures, the width of vector registers has increased from 128 to 256 bits in modern processors with AVX support and 512 bits in CPUs supporting AVX-512 instruction set. This trend of increasing size of the register file is expected to continue into future architectures, thus the potential gain of vectorization is not negligible. Assuming a 256-bit wide register, a fully vectorized implementation is potentially 4 or $8\times$ faster for double and single precision floating points respectively. The lack of SIMDization can impact performance significantly. Figure 4 shows that without SIMD, we lose 75% of peak performance on all three architectures.

To achieve better code portability for architectures with different vector register sizes, we choose a compiler generated

vectorized code over manual SIMDization using intrinsics. However, the compiler initially failed to auto-vectorize the code, for the most part. To aid and enable the compiler to auto-vectorize the solver, we implement a number of optimizations and transformations. In this section, we discuss these optimizations which are categorized as code structure and data structure transformations. We use the Intel compiler and its vectorization guide [1] while applying these optimizations.

1) SIMD-aware Code Structure: The transformations described in this section reorganize the code structure to enable auto-vectorization and also improve the performance gain of vectorization.

a) Loop Unswitching: In this modification, we avoid using `if/else` conditional statements within the body of the loops. Even though the compiler could potentially vectorize if such statements can be implemented as masked assignments [1], the performance gain is considerably higher without these conditionals. In order to remove the `if/else` clauses, we take either of the following approaches, depending on the clause – a) move the statement outside the loop and duplicate the body of the loop, or b) use C++ conditional operator.

b) Loop Fission: This transformation is largely beneficial when vectorization fails due to dependencies. For instance, in cases where a value is updated, used and then updated again, the compiler detects a **Write After Write** dependency (Intel compiler calls this an Output dependency) and fails to vectorize. The code in Loop 1 is an example of such a case.

Loop 1: Sample loop with dependencies

```
for (int i = 0; i = max_i; ++i) {
  A[i] = SomeValue;
  B[i] = /*Some calculation on A[i];*/
  A[i] = UpdatedValue;
}
```

Breaking this loop into two separate loops as shown in Loop 2, allows the compiler to vectorize both loops since there are no dependencies within a loop.

Loop 2: An example of loop fission

```
for (int i = 0; i = max_i; ++i)
  A[i] = SomeValue;
for (int i = 0; i = max_i; ++i) {
  B[i] = /*Some calculation on A[i];*/
  A[i] = UpdatedValue;
}
```

This pattern occurs in viscous flux computation where we calculate and then use values such as average velocity gradients. While this adds to the overhead of a new loop, this cost is negligible compared to the performance gain due to vectorization.

c) Loop Unrolling: Loop unrolling is a simple transformation and the compiler is able to apply this optimization to most loops. However, in the case of nested loops, the compiler only vectorizes the innermost loop. This is not efficient in cases where the inner loop has iteration counts. In such cases, we manually unroll the small loops in order to force the compiler to vectorize a larger loop.

2) SIMD-aware Data Layout: While the preceding code structure modifications enable vectorization, we further improve the performance gain by re-designing the data structures layout in memory in a way that better suits SIMD instructions.

a) Pointer Aliasing: Using the *__restrict__* keyword in C++ for pointer declaration indicates that the target of that pointer will not be accessed by any other pointers, thus avoiding pointer aliasing. Even though using this keyword requires extra care while working with pointers, it is essential for ensuring auto-vectorization.

b) Structure of Arrays: Table III summarizes the variables used in this solver. Inviscid fluxes, artificial dissipation, and viscous fluxes are stored as a vector of size 5 for each cell since these values are computed for all the conservative variables as discussed in Section II-A. These values can be stored in memory as either an Array of Structures (AoS) or a Structure of Arrays (SoA). The advantage of using AoS is that it better exploits locality when accessing the 5 components for each cell in a loop since all 5 values can be read into the cache in a cache line. However, the disadvantage of this approach is that it requires non-unit stride memory accesses to fetch different components of neighboring cells into vector registers. In contrast, SoA is better suited for vectorization as it allows for unit stride load and store operations within the innermost loop over the grid. Thus to achieve better performance with SIMDization, we reorganize all the aforementioned data structures as SoA's, where each array holds the values for one of the conservative variables.

The performance improvement with the code structure and data-layout transformations is shown as *SIMD transformations* in Figure 5. The red bar shows the speedup achieved with compiler auto-vectorization after applying these transformations. Combined, we achieve speedups of $2.3 - 3.7\times$ on Haswell, $1.5-3.1\times$ on Abu Dhabi, and $1.6-2.3\times$ on Broadwell respectively. Note that the speedup due to vectorization decreases as we increase the number of threads on all three systems. This is because the code becomes progressively more memory-bound as we increase the number of threads and as a result, the benefit of vectorization decreases. On the contrary, the performance improvement due to blocking increases as we scale the number of threads for the same reason.

V. Comparison with DSL

In this section, we attempt to answer two key questions – (a) Can CFD applications be expressed in stencil DSLs? (b) Can such an implementation deliver a sufficient combination of optimizations to compete with a hand-tuned code? If not, what are its limitations?

In order to answer the above questions, we choose Halide [15], a DSL designed for image processing pipelines which consist of multi-stencils. It decouples the algorithm specification from the schedule that can be further tuned either manually or automatically for performance. An advantage of Halide is that the scheduling schemes account not only

for a single stencil but also multi-stencils and how they operate together. This is essential for optimal scheduling of an application such as our CFD solver. Halide is one of the DSLs that is suitable for such multi-stencil applications and this was the main reason behind our choice of Halide for comparison.

Halide provides knobs for various optimizations such as tiling, parallelization, and vectorization. The main challenge is efficiently applying these optimizations and finding the schedule that achieves the best performance. The optimized Halide schedule we obtained is similar to our *intra-stencil* fusion scheduling discussed in Section IV, where we find the best ordering for computing the intermediate values and find the optimal trade-off between locality, redundant computation, and available parallelism. Finding the best schedule in Halide also entails finding the best tiling size in both x and y directions, similar to finding the best block size in our hand-tuned code. While porting the solver to Halide required minimal effort, finding the optimal schedule was non-trivial. We compare the speedup achieved by our hand-tuned code against our best schedule in Halide. Table IV summarizes these results.

Across the board, we achieve higher speedups after applying single-core optimizations compared to Halide. While Halide supports optimizations such as loop unrolling, loop fusion, tiling in x and y dimensions, it does not support strength reduction. Also, it has the additional cost of estimating the bounds for all the stencil loop computations which adds to the overhead.

We then apply vectorization on top of single-core optimizations. Halide does not gain much from vectorization, mainly because SIMDization is not one of the optimizations focused on in the context of image processing. Additionally, our hand-tuned implementation includes data layout transformations to enable better vectorization which is not done by Halide.

Finally, we apply parallelization on top of vectorization. Halide currently does not account for NUMA (adding support for NUMA is in progress [6]) and therefore we achieve better scalability when using all the cores. This is especially evident in Abu Dhabi that has the most number of NUMA nodes and there is a significant difference in speedup achieved with our NUMA-aware OpenMP implementation compared to Halide.

It is worth noting that Halide delivers the best performance for stencil patterns that are categorized as *cell-centered stencils* in Section II-B. These are the commonly studied stencil patterns in most of the stencil optimization studies. This is one of the reasons for the poor performance of Halide and other similar DSLs on real applications. We also compare our manual Halide schedule against the schedule generated by the auto scheduler [13]. Our optimized schedule performs $2-20\times$ better than the auto scheduler for different stencil patterns, similarly showing best performance for cell-centered stencils.

VI. Related work

Stencil calculations perform sweeps through an entire data structure that is typically larger than the capacity of the data cache. As a result, these computations achieve a low fraction of theoretical peak performance. This has lead to a vast number of studies that focus on tiling optimizations to exploit locality by performing operations on cache-sized blocks of data [4], [5], [10], [17], [23]. The preceding work mostly focuses on 2nd order stencils, whereas higher order stencils tend to become more compute bound. [2] implements a partial sum optimization that makes higher-order stencils bandwidth bound, prior to applying similar optimizations that improve the performance of memory-bound stencils.

While most traditional tiling optimizations use domain decomposition to improve spatial locality, studies have also focused on exploiting the locality in the time dimension [7], [19], [22], [25]. An extension to single time step blocking is the time-skewing algorithm [19], [25] which blocks in both space and time.

Another body of work in this domain is on developing domain-specific compilers and code generators that target stencil computations. Halide [13], [15] focuses on image processing pipelines, decouples algorithm and schedule, and provides an auto-scheduler for finding the optimal schedule. Pochoir [20] uses cache-oblivious tiling. [2] focuses on compiler-directed polyhedral transformations of stencils but for higher-order stencils that are typically compute-bound.

This work that captures multi-physics flows differs from the preceding work in a couple of ways. First, we optimize a numerically accurate solver that capture real physics and consist of multiple terms, and therefore different stencil calculations in each time step. Optimizing these different stencils separately and also as an entire solver that requires scheduling of the different stencils is a challenge addressed in our work. Halide is one of the few compiler frameworks that consider the trade-offs of scheduling multiple stencils. We present a comparison against Halide and suggest optimizations to bridge the gap between hand-tuned code and code generated by DSL's.

While there have been efforts in mapping CFD solvers to multicore architectures, most of the work in this field has focussed on a parallel implementation using either OpenMP or MPI [18], as opposed to our work where we focus on both efficient parallelization and multi-stencil specific optimizations.

VII. Conclusion

Given that single-node multicore performance will be critical to scalability on next-generation extreme scale systems, we believe our extensive study of optimizations and parallelization contributes a solid foundation for the future scalable software infrastructure for next-generation CFD on such machines. We use the roofline model to guide the optimizations to improve the performance based on the arithmetic intensity at each step and push the solver towards the compute-bound region. The optimizations discussed in this work were designed not only to improve the execution time but also to reach a higher percentage of the peak performance of the machine.

While some of the optimizations discussed in this work have been previously applied to stencil computations [4], [5], [7], [14], this is one of the first efforts in addressing the challenges of an entire solver which consists of multiple stencils. We discuss the different patterns of these stencils and

TABLE IV: Comparison of the speedup achieved by the hand-tuned code against Halide. Optimization stands for strength reduction, fusion, and blocking. +Vectorization is SIMD-aware transformations and compiler auto-vectorization on top of single-core optimizations, and +Parallelization is parallelization on top of vectorization.

	Haswell		Abu Dhabi		Broadwell	
	Hand-tuned	Halide opt	Hand-tuned	Halide opt	Hand-tuned	Halide opt
Optimization	3.5×	1.5×	3.0×	1.3×	3.2×	1.4×
+ Vectorization	3.6×	1.1×	2.3×	1.0×	2.8×	1.2×
+ Parallelization	7.9×	5.8×	23.3×	5.1×	17.6×	6.2×

the challenges they give rise to, along with an in-depth analysis of an important real-world application.

Furthermore, we implement this solver using a state-of-the-art domain-specific language (Halide) to compare and quantify the performance achieved by DSLs against hand-tuned codes for this application domain. While porting this solver to Halide was relatively easy, finding the optimal schedule was challenging and required an understanding of the underlying architecture. Our solver outperforms Halide by 10×, 24× and 15× on the three different platforms.

Looking forward, we see numerous opportunities for improving stencil DSLs. First and foremost, Halide and other stencil DSLs are largely focused on optimizing cell-centered stencils. While this is an important stencil pattern, real applications consist of multi-stencils which are cell-centered, face-centered, and vertex-centered. In-depth analysis and optimization across these multi-stencils with different characteristics should be a priority for future stencil DSLs to bridge the performance gap for real CFD applications. Next, NUMA-aware data allocation is critical on current and future systems. Halide's poor scaling with increasing number of cores is due to lack of support for NUMA and lack of efficient vectorization or SIMD-aware data layout transformations. We believe addressing the above deficiencies will make stencil DSLs competitive with hand-tuned codes.

REFERENCES

[1] "A Guide to Auto-vectorization with Intel® C++ Compilers," https://software.intel.com/en-us/articles/a-guide-to-auto-vectorization-with-intel-c-compilers, 2012.

[2] P. Basu, M. Hall, S. Williams, B. V. Straalen, L. Oliker, and P. Colella, "Compiler-Directed Transformation for Higher-Order Stencils," in *Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2015, pp. 313–323.

[3] J. Cocke and K. Kennedy, "An Algorithm for Reduction of Operator Strength," *Comm. of the ACM*, vol. 20, no. 11, pp. 850–856, 1977.

[4] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors," *SIAM Review*, vol. 51, pp. 129–159, 2009.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures," in *Proceedings of the ACM/IEEE Conf. on Supercomputing*, Piscataway, NJ, USA, 2008.

[6] T. Denniston, S. Kamil, and S. Amarasinghe, "Distributed Halide," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, NY, USA, 2016.

[7] M. Frigo and V. Strumpen, "Cache Oblivious Stencil Computations," in *Proceedings of the Intl. Conf. on Supercomputing*, New York, NY, 2005.

[8] A. Jameson, "Time-Dependent Calculations Using Multigrid, with Applications to Unsteady Flows Past Airfoils and Wings," *AIAA 91-1596*, 1991.

[9] A. Jameson, W. Schmidt, and E. Turkel, "Numerical Solution of the Euler Equations by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes," *AIAA Paper 81-1259*, 1981.

[10] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Implicit and Explicit Optimizations for Stencil Computations," in *Proceedings of the Workshop on Memory System Performance and Correctness*. New York, NY, USA: ACM, 2006.

[11] F. Liu and X. Zheng, "A Strongly-Coupled Time-Marching Method for Solving the Navier-Stokes and $k-\omega$ Turbulence Model Equations with Multigrid," *Journal of Computational Physics*, vol. 128, no. 2, pp. 289–300, 1996.

[12] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "PAPI: A Portable Interface to Hardware Performance Counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[13] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, "Automatically scheduling halide image processing pipelines," *ACM Trans. Graph.*, vol. 35, no. 4, pp. 83:1–83:11, Jul. 2016.

[14] A. Nguyen, N. Satish, J. Chhugani, C. Kim, and P. Dubey, "3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–13.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," in *Proceedings of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation*, New York, NY, USA, 2013.

[16] J. Reinders, *VTune (TM) Performance Analyzer Essentials: Measurement and Tuning Techniques for Software Developers*. Intel Press, 2004.

[17] G. Rivera and C.-W. Tseng, "Tiling Optimizations for 3D Scientific Computations," in *Proceedings of the 2000 ACM/IEEE Conf. on Supercomputing*, Washington, DC, USA, 2000.

[18] C. Simmendinger, J. Jagerskupper, and R. Machado, "A PGAS-based implementation for the unstructured CFD solver TAU," *In: 5th Conference on Partitioned Global Address Space Programing Models, Tremont House, Galveston Island*, 2011.

[19] Y. Song and Z. Li, "New Tiling Techniques to Improve Cache Temporal Locality," *SIGPLAN Not.*, vol. 34, no. 5, pp. 215–228, May 1999.

[20] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson, "The Pochoir Stencil Compiler," in *Proceedings of the ACM symp. on Parallelism in algorithms and architectures*, 2011.

[21] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments," in *Proceedings of the Int'l Conf. on Parallel Processing Workshops*, 2010.

[22] J. Treibig, G. Wellein, and G. Hager, "Efficient multicore-aware parallelization strategies for iterative stencil computations," *Journal of Computational Science*, vol. 2, no. 2, pp. 130 – 137, 2011.

[23] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–14.

[24] S. Williams, A. Waterman, and D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[25] D. Wonnacott, "Using Time Skewing to Eliminate Idle Time due to Memory Bandwidth and Network Limitations," in *Parallel and Distributed Processing Symposium, 2000.*, 2000, pp. 171–180.

[26] J. Xiong, P. Nielsen, F. Liu, and D. Papamoschou, "Computation of High-Speed Coaxial Jets with Fan Flow Deflection," *AIAA Journal*, vol. 48, no. 10, pp. 2249–2262, 2010.